LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

UCRL-CONF-209840

DynTG: A tool for Interactive, Dynamic Instrumentation

M. Schulz, J. May, J. Gyllenhaal

February 18, 2005

Tools for Program Development and Analysis in Computational Science
Atlanta, GA, United States
May 22, 2005 through May 25, 2005

DynTG: A Tool for Interactive, Dynamic Instrumentation

Martin Schulz, John May, and John Gyllenhaal
{*schulzm,johnmay,gyllen*}*@llnl.gov*

Center for Applied Scientific Computing
Lawrence Livermore National Laboratory

Abstract. With the increasing complexity of today's systems, detailed performance analysis is more important than ever. We have developed DynTG, a tool for interactive, dynamic instrumentation. It uses performance module plugins to reconfigure the data acquisition and provides a source browser that allows users to insert any probe functionality provided by the modules dynamically into the target application. Any instrumentation can be added both before and during the application's execution and the acquired data is presented in realtime within the source viewer. This enables users to monitor their applications' progress and interactively control and adapt the instrumentation based on their observations.

1 Motivation

Users require comprehensive performance toolkits to analyze their applications, find bottlenecks, and derive appropriate optimizations. Many research projects have focused on this problem and have developed a large range of such tools. Some of the more prominent examples are TAU/ParaProf [1], SvPablo [4], and Kojak [9]. Also several commercial tools are available, including Intel's Vtune and SGI's SpeedShop. All of them allow users to gather a large variety of performance metrics and often provide graphical user interfaces to present the results. However, most of them rely on static source code instrumentation and/or postmortem performance analysis and hence lack interactive capabilities.

Dynamic instrumentation is the first step towards interactive tools. Most tools in this area are built on top of DynInst [2], an API to dynamically insert arbitrary code snippets into running applications. Performance monitoring infrastructures, like OMIS [6] or DPCL [3], and tools like Paradyn [8] use this mechanism to insert performance probes into the application. However, these tools don't provide the user with interactive, source-code based environments to control their instrumentation.

With DynTG, we have developed a tool that integrates dynamic instrumentation with a source browser and provides users with a fully interactive performance analysis environment. It builds on top of DPCL to dynamically instrument applications and to acquire realtime performance data. The actual performance probes are dynamically loaded into DynTG using a modular plugin concept. By providing new performance modules, users can reconfigure DynTG

This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48. UCRL-CONF-209840.

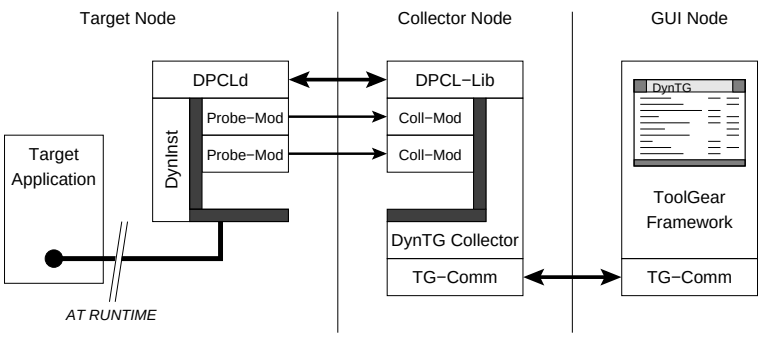

Fig. 1. Architecture of DynTG.

to suite their particular needs and extend it with new performance metrics and data sources.

All instrumentation is controlled through a scalable source browser developed using the Tool Gear framework [7]. Performance modules dynamically register their data sources and instrumentation actions with the Tool Gear based GUI and thereby dynamically customize the tool based on their requirements. Further, all performance data gathered by the probe modules is transmitted to the GUI and displayed inside the source browser in realtime. This allows users to observe changes in the application behavior, detect bottlenecks, and, if necessary, add or remove instrumentation at runtime.

2 The DynTG Architecture

A tool for interactive, dynamic instrumentation must a) provide an extensive and extensible set of data probes; and b) allow users to insert instrumentation into their codes through a source browser. DynTG achieves these two goals by a) providing a plugin concept using dynamically loaded performance modules; and b) relying on and extending Tool Gear [7], a versatile framework for building graphical tools. DynTG enables users to specify instrumentation locations directly in the source code and to select appropriate data probes at these locations. Once selected, DynTG uses DPCL [3]/DynInst [2] to instrument the target application with the selected probe functionality. The resulting performance data is returned to the GUI and displayed within the source browser on a per-line basis.

Figure 1 shows the basic architecture of DynTG. It is divided into three major components — performance probes, the collector, and the GUI. The GUI allows the user to spawn and control the collector, which acts as a link or proxy between the application and the GUI. Once started, it uses DPCL to launch and control the application on the target system and to install the probe modules. These modules return any acquired performance data through DPCL to corresponding collector modules. There, the performance data is preprocessed and then forwarded through the collector infrastructure to the GUI for display using the Tool Gear communication mechanism.

Since both libraries use standard sockets for the actual communication, we can (if necessary) place each component onto a different node. For example, the GUI can run on the user's desktop and communicate with a remote collector on a front-end node, which then interacts with the probe modules on the application's compute nodes. This matches the structure of typical cluster or computer center configurations and has the ability to provide a fast, interactive GUI, while enabling online accesses to running applications on remote systems.

3 Probe/Collector Modules

DynTG has the ability to dynamically load and activate performance modules. This allows users to extend, configure, and customize the tool according the specific monitoring needs.

3.1 Module Concept

Each module is implemented in two separate parts: a probe module, which is dynamically inserted into the running application; and a collector module, which is loaded into the DynTG collector.

Probe modules are generally written in C and implement custom data probes. They communicate the acquired results back to the corresponding collector module using DPCL's transfer mechanisms. In most cases this transfer is accomplished with a single call to DPCL's *Ais_send*.

The corresponding collector module implements the necessary DPCL callback, which receives the data, if necessary transforms it into an appropriate format, potentially performs a preliminary analysis, and then hands the data to the GUI. In addition, the collector module implements two management routines required by the framework: a query routine, which returns a list of requirements for this modules along with a short textual description of its functionality, and an initialization routine.

The module interfaces are kept small by moving common and low-level functionality into the framework or the communication libraries. This simplifies the module development process, lowers the learning curve, and hence allows even unexperienced users not familiar with DPCL or Tool Gear to develop new probe/collector modules in a short amount of time and thereby to dynamically extend DynTG without changing the source of either collector or GUI.

3.2 Installation and Activation

During startup, the collector scans a module library and queries each module found for its installation requirements using the query routine introduced above. Based on the results it excludes incompatible modules (e.g., those that rely on specialized hardware or software) and then loads and activates the remaining ones. During this activation, the module locates the corresponding probe module and links it to a DPCL callback.

After all probes are loaded and initiated, the collector registers the module's functionality with Tool Gear: it compiles a list of possible actions (e.g., potential instrumentation insertion and removal events) and data sources, and then sends this list to the GUI. Together with this registration, it establishes a name space, which is later used to react on GUI actions or send observed data events.

3.3 Module Selection

It is not always necessary or beneficial to load and activate all available instrumentation modules. Each module consumes resources and also might incur conflicts with other modules. This is especially critical with an increasing number of modules. DynTG therefore enables the user to select and activate a subset of all available modules. This can either be done using a command line parameter during the collector startup or interactively through the DynTG GUI. For the latter, the collector forwards a list of all available modules together with the short description returned by the above mentioned query routine to the GUI. The GUI then builds a selection dialog (see Figure 2/right,bottom) and queries the user for input. Based on the user's decision, the GUI requests the activation of the selected modules in the DynTG collector.

3.4 Existing Modules

Currently, we have implemented three probe/collector modules:

Counter: The counter module provides a counter that counts how often a program executes a specific location. This module can be used to examine iteration counts, distribution of execution paths on conditionals, or callgraph coverage.

Timer: The timer module provides wall clock timing using the UNIX *gettimeofday* call. The timer is controlled by two actions that the user can instrument the code with: start timer and stop timer. A stop timer action thereby always corresponds to the most recently started active timer and computes the time difference between the two. In order to keep track of these correspondences, the timer module maintains a timer stack: during a start timer action, the current time value is pushed onto the stack and during a stop, this value is retrieved from the stack and used to compute the time difference.

PMAPI: This module provides direct access the hardware counters on IBM Power systems using AIX's Performance Monitoring API (PMAPI). Due to hardware constraints only a subset of events can be accessed at the same time. To further complicate things, each of the eight hardware counters available in Power-3 or Power-4 systems can only be used for a specific and distinctly different set of events. During the loadtime of the PMAPI collector module, the user specifies a set of events of interest in a priority list[1]. The module then computes an optimal mapping of this list to the available hardware resources on the target architecture and configures the hardware counters appropriately. Once completed, the PMAPI module registers its start and stop events for each available and configured counter and maintains a stack, similar to the one in the timer module, to associate the corresponding start and stop events.

Table 1 shows the number of lines used for the implementation of each module (including comments). Despite the fact that lines of codes are naturally a very inaccurate measurement, the numbers show that the implementation complexity for each module is low. The collector modules show a constant overhead needed

[1] Currently this list is specified in a configuration file; we are currently working on a GUI version of this selection process

Module	Probe	Collector
Counter	10	157
Timer	170	171
PMAPI (+ shared utility)	99 (+199)	213 (+199)

Table 1. Implementation complexity of probe/collector modules in lines of code.

to load and register the module, while the complexity in the probe module only depends on the complexity of the actual probe functionality.

4 GUI Support for Interactive Instrumentation

Dynamic instrumentation enables users to selectively activate instrumentation in their codes. In order to fully take advantage of this feature, users must be able to interactively browse their source code, identify possible instrumentation locations, and interactively choose the required instrumentation at those points. DynTG uses Tool Gear [7] to implement this functionality and extends it to provide reconfigurable instrumentation and support for modules.

4.1 Integrating DynTG with the Tool Gear Framework

Source code viewers are complex and difficult to implement, yet required by a large number of tools. The Tool Gear project [7] was born based on this observation and provides a versatile framework for the development of interactive tools. As its core it provides a scalable source viewer, which can display both the source code itself as well as multiple performance metrics on a per source line basis. The performance data itself is stored inside Tool Gear using a database. Data can directly be added to this database from a collector using the Tool Gear communication library.

The Tool Gear source browser provides a collapsible view of all source modules on the left side of the window and displays the corresponding performance information in column style for each source line on the right of it. We use this generic setup to display performance data acquired by the probe modules. Each module can register one or more data sources and each of these sources will be displayed using a separate column within the source browser. The DynTG collector continuously updates these performance observations during the program's runtime. This enables the user to monitor the progress of their code, spot potential bottlenecks early, and react by inserting additional or removing unnecessary instrumentation. In addition, Tool Gear aggregates the presented performance data hierarchically showing the performance of a complete function, module, or application.

Figure 2 (left) shows a typical view of the source code browser as it is used by DynTG. In this example we use the timer probe/collector module to instrument a simple test application and to measure wall clock times of selected function invocations within *main*. We place start timer instrumentations (green icons) before and end time instrumentations (red icons) after the relevant function calls. The results are displayed in a single data column on the right and are

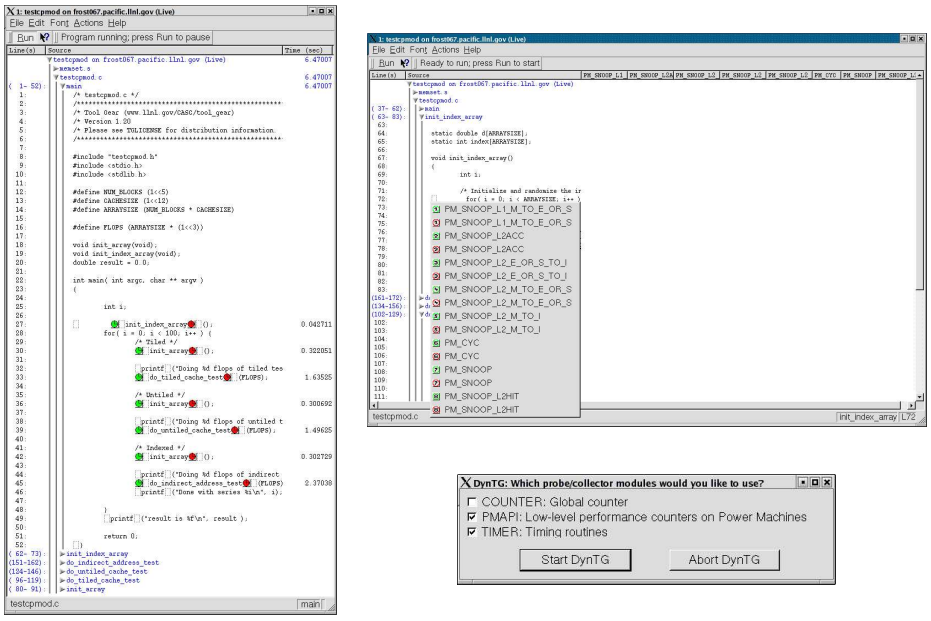

Fig. 2. DynTG GUI: source browser with timer experiment (left), context-sensitive action menu for PMAPI module (right,top), module selection dialog (right,bottom).

updated every time the application executes one of the instrumentation points. In addition, the top rows show the accumulated runtime of all counters in the main routine, the *testcpmod.c* module, and the total application.

4.2 DynTG Startup and Configuration

During the startup of DynTG, the Tool Gear infrastructure launches the DynTG collector on the remote node and establishes the communication using its communication library. As described in Section 3.2, the collector then searches for all available modules and activates a user-selected subset of those modules. During their activation, the module compiles a list of actions provided by this module and forwards this to the GUI. Examples for this can be "Start a timer" or "Count at this location". When the GUI receives a request for a new action, it selects an appropriate icon and adds it to the list of available actions.

This runtime configuration process guarantees that the GUI is independent from the collector: the GUI itself does not enable or provide any functionality to the user. Instead, all actions required by a collector are defined and installed by the collector. As a consequence, new modules can easily be added without modifying the GUI code itself. The only exception are probes that require new datatypes for the performance database or new visualization or display modes. Most performance modules, however, are able to work with the existing infrastructure.

In addition, the GUI queries possible instrumentation points from the collector at startup and marks them in the source display. At these locations DynTG

Benchmark	Absolute runtime				Overhead		
	Baseline	Counter	Timer	PMAPI	Counter	Timer	PMAPI
SMG2000	56.6s	57.7s	58.0s	58.0s	1.94%	2.47%	2.47%
sPPM	53.4s	56.7s	59.5s	60.0s	6.18%	11.4%	12.4%

Table 2. Runtime and Overhead measurements of the three existing DynTG modules.

offers a context-sensitive instrumentation menu that shows all possible instrumentation actions that can be inserted or removed at that particular location. The user can then select from this menu and insert the instrumentation. In addition, the DynTG includes a bulk instrumentation mechanism to instrument a set of routines. For this features, the user selects the instrumentation action as above and specifies the name of the routines to be instrumentation using regular expressions.

Any instrumentation can be done both before the program's execution (in which case the performance data from the whole run is captured) or at runtime (in which case the performance data is gathered from the time of the instrumentation). It is further possible to dynamically remove instrumentation points.

Figure 2 (right,top) shows an example of such an instrumentation menu. In this example, the PMAPI module is loaded and instantiated with eight different native hardware counters. Consequently the menu has 16 entries: a start (green icons) and a stop (red icons) event for each counter type. Further, the GUI displays eight columns for the performance data, again one for each hardware counter. In both cases, the collector has queried the counter names from PMAPI and registered them with the GUI dynamically.

5 Experiments

In the following section we present experimental results quantifying the overhead caused by DynTG. All experiments were conducted on one CPU of a dedicated 16-way IBM Power-3 system. We use two numerical applications from the ASC Purple Benchmark Codes [5]: SMG2000 with a working set of 60x60x60 and sPPM with a working set of 64x64x64. We inserted a single instrumentation action into a central location and measured the overhead. This location was executed 34167 times in SMG2000 and 245760 times in sPPM.

As the results in Table 2 show, DynTG has an low impact on performance even if the instrumentation is executed very frequently like in sPPM (over 4000 times per second). In general, the overhead corresponds to the work performed within the data probes inserted into the application. Consequently, the overhead of the *Timer* and *PMAPI* modules is higher, since these modules require system calls to gather their data. In addition, we compared the runtime of an application run under the control of DynTG without instrumentation to the baseline and observed no difference in execution time.

6 Conclusions

In this paper, we presented DynTG, a tool for interactive, dynamic instrumentation. It includes a scalable source browser, in which the user can view all possible

instrumentation points and insert probes into the application. Once the probes are installed, the performance data gathered by them is displayed in realtime inside the source browser at the location of the instrumentation point.

DynTG uses DPCL to realize dynamic instrumentation and relies on and extends Tool Gear to provide the scalable source browser. To connect those two components DynTG implements a reconfigurable collector that acts as a proxy between the data probes and the GUI. The probes as well as the probe specific parts of the collector are implemented as dynamically loadable plugin modules. This concept enables users to customize and reconfigure the tool depending on their actual monitoring requirements.

References

1. R. Bell, A. Malony, and S. Shende. A Portable, Extensible, and Scalable Tool for Parallel Performance Profile Analysis. In *Proceedings of the International Conference on Parallel and Distributed Computing (Euro-Par 2003)*, pages 17–26, Aug. 2003.
2. B. Buck and J. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, 2000.
3. L. DeRose, T. Hoover, and J. Hollingsworth. The Dynamic Probe Class Library — An Infrastructure for Developing Instrumentation for Performance Tools. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium*, Apr. 2001.
4. L. DeRose and D. Reed. SvPablo: A Multi-Language Architecture-Independent Performance Analysis System. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, Sept. 1999.
5. Lawrence Livermore National Laboratory. The ASCI purple benchmark codes. http://www.llnl.gov/asci/purple/benchmarks/limited/code_list.html, Oct. 2002.
6. T. Ludwig, R. Wismüller, V. Sunderam, and A. Bode. *OMIS — On-line Monitoring Interface Specification (Version 2.0)*, volume 9 of *LRR-TUM Research Report Series*. Shaker Verlag, Aachen, Germany, 1997. ISBN 3-8265-3035-7.
7. J. May and J. Gyllenhaal. Tool gear: Infrastructure for parallel tools. In *Proceedings of the 2003 International Conference on Parallel and Distributed Techniques and Applications*, June 2003.
8. B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, R. Irvin, K. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn Parallel Performance Measurement Tool. *IEEE Computer*, 28(11):37–46, Nov. 1995.
9. B. Mohr and F. Wolf. KOJAK - A Tool Set for Automatic Performance Analysis of Parallel Programs. In *Proceedings of the International Conference on Parallel and Distributed Computing (Euro-Par 2003)*, pages 1301–1304, Aug. 2003.